1

WRDC-TR-90-8007
Volume III
Part 9

# INTEGRATED INFORMATION SUPPORT SYSTEM (IISS)
Volume III - Configuration Management
Part 9 - Software Development Guidelines

J. Maxwell

Control Data Corporation
Integration Technology Services
2970 Presidential Drive
Fairborn, OH 45324-6209

September 1990

Final Report for Period 1 April 1987 - 31 December 1990

MANUFACTURING TECHNOLOGY DIRECTORATE
WRIGHT RESEARCH AND DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6533

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever, regardless whether or not the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data. It should not, therefore, be construed or implied by any person, persons, or organization that the Government is licensing or conveying any rights or permission to manufacture, use, or market any patented invention that may in any way be related thereto.

This technical report has been reviewed and is approved for publication.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations

[illegible]

DAVID L. JUDSON, Project Manager
WRDC/MTI
Wright-Patterson AFB, OH 45433-6533

25 July 91
DATE

FOR THE COMMANDER:

Bruce Rasmussen

BRUCE A. RASMUSSEN, Chief
WRDC/MTI
Wright-Patterson AFB, OH 45433-6533

25 July 91
DATE

If your address has changed, if you wish to be removed form our mailing list, or if the addressee is no longer employed by your organization please notify WRDC/MTI, Wright-Patterson Air Force Base, OH 45433-6533 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for Public Release;<br>Distribution is Unlimited. |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>UM 620324000 | 5. MONITORING ORGANIZATION REPORT NUMBER(S)<br>WRDC-TR-90-8007 Vol. III, Part 9 |
| 6a. NAME OF PERFORMING ORGANIZATION<br>Control Data Corporation;<br>Integration Technology Services<br>6b. OFFICE SYMBOL (if applicable) | 7a. NAME OF MONITORING ORGANIZATION<br>WRDC/MTI |
| 6c. ADDRESS (City, State, and ZIP Code)<br>2970 Presidential Drive<br>Fairborn, OH 45324-6209 | 7b. ADDRESS (City, State, and ZIP Code)<br>WPAFB, OH 45433-6533 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION<br>Wright Research and Development Center,<br>Air Force Systems Command, USAF<br>8b. OFFICE SYMBOL (if applicable)<br>WRDC/MTI | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUM.<br>F33600-87-C-0464 |
| 8c. ADDRESS (City, State, and ZIP Code)<br>Wright-P[illegible]erson AFB, Ohio 45433-6533 | 10. SOURCE OF FUNDING NOS. |
| 11. TITLE (Include Security Classification)<br>See Block 19 | PROGRAM ELEMENT NO. 78011F; PROJECT NO. 595600; TASK NO. F95600; WORK UNIT NO. 20950607 |

12. PERSONAL AUTHOR(S)
Control Data Corporation: Maxwell, J.

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Yr.,Mo.,Day) | 15. PAGE COUNT |
|---|---|---|---|
| Final Report | 4/1/87-12/31/90 | 1990 September 30 | 51 |

16. SUPPLEMENTARY NOTATION

WRDC/MTI Project Priority 6203

17. COSATI CODES

| FIELD | GROUP | SUB GR. |
|---|---|---|
| 1308 | 0905 | |

18. SUBJECT TERMS (Continue on reverse if necessary and identify block no.)

19. ABSTRACT (Continue on reverse if necessary and identify block number)

This document identifies and explains the guidelines and conventions used by IISS testbed developers throughout the IISS software development life cycle.

Block 11 - INTEGRATED INFORMATION SUPPORT SYSTEM (IISS)
Vol III - Configuration Management
Part 9 - Software Development Guidelines

| | |
|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT<br>UNCLASSIFIED/UNLIMITED x SAME AS RPT. DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL<br>David L. Judson | 22b. TELEPHONE NO. (Include Area Code) (513) 255-7371 — 22c. OFFICE SYMBOL WRDC/MTI |

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

UM 620324000
30 September 1990

## FOREWORD

This technical report covers work performed under Air Force Contract F33600-87-C-0464, DAPro Project. This contract is sponsored by the Manufacturing Technology Directorate, Air Force Systems Command, Wright-Patterson Air Force Base, Ohio. It was administered under the technical direction of Mr. Bruce A. Rasmussen, Branch Chief, Integration Technology Division, Manufacturing Technology Directorate, through Mr. David L. Judson, Project Manager. The Prime Contractor was Integration Technology Services, Software Programs Division, of the Control Data Corporation, Dayton, Ohio, under the direction of Mr. W. A. Osborne. The DAPro Project Manager for Control Data Corporation was Mr. Jimmy P. Maxwell.

The DAPro project was created to continue the development, test, and demonstration of the Integrated Information Support System (IISS). The IISS technology work comprises enhancements to IISS software and the establishment and operation of IISS test bed hardware and communications for developers and users.

The following list names the Control Data Corporation subcontractors and their contributing activities:

| SUBCONTRACTOR | ROLE |
|---|---|
| Control Data Corporation | Responsible for the overall Common Data Model design development and implementation, IISS integration and test, and technology transfer of IISS. |
| D. Appleton Company | Responsible for providing software information services for the Common Data Model and IDEF1X integration methodology. |
| ONTEK | Responsible for defining and testing a representative integrated system base in Artificial Intelligence techniques to establish fitness for use. |
| Simpact Corporation | Responsible for Communication development. |

| | |
|---|---|
| Structural Dynamics Research Corporation | Responsible for User Interfaces, Virtual Terminal Interface,and Network Transaction Manager design, development, implementation, and support. |
| Arizona State University | Responsible for test bed operations and support. |

## Table of Contents

| | | Page |
|---|---|---|
| SECTION 1.0 | INTRODUCTION | 1-1 |
| SECTION 2.0 | PRELIMINARY DESIGN GUIDELINES/ CONVENTIONS | 2-1 |
| 2.1 | Functional Requirements | 2-1 |
| 2.2 | Information Requirements | 2-1 |
| 2.3 | IDEF1 Traceability | 2-1 |
| SECTION 3.0 | DETAILED DESIGN GUIDELINES | 3-1 |
| 3.1 | Module Specification | 3-1 |
| 3.2 | Procedural Descriptions | 3-2 |
| 3.2.1 | Primary Purpose | 3-2 |
| 3.2.2 | Level of Detail | 3-2 |
| 3.2.3 | Basis for Development | 3-2 |
| 3.2.4 | Syntactical Rules | 3-2 |
| 3.3 | Suggestions for Complexity Adding | 3-2 |
| 3.3.1 | Module Number | 3-2 |
| 3.3.2 | Interactions | 3-2 |
| SECTION 4.0 | PROGRAM CONSTRUCTION GUIDELINES/CONVENTIONS | 4-1 |
| 4.1 | General Program Construction Guidelines/Conventions | 4-1 |
| 4.1.1 | Design Documents Kept Current | 4-1 |
| 4.1.2 | Program Banner | 4-1 |
| 4.1.3 | Program Header | 4-1 |
| 4.1.4 | Special Comments | 4-1 |
| 4.1.5 | Error Handling | 4-2 |
| 4.1.6 | Mesage Storage | 4-2 |
| 4.1.7 | Tables | 4-2 |
| 4.1.8 | Indentations | 4-2 |
| 4.1.9 | Nested IF Statements | 4-2 |
| 4.2 | COBOL Language Guidelines/Conventions | 4-2 |
| 4.3 | FORTRAN Language Guidelines/Conventions | 4-3 |
| 4.4 | "C" Language Guidelines/Conventions | 4-4 |
| 4.4.1 | Portability Issues | 4-4 |
| 4.4.1.1 | Data Types | 4-4 |
| 4.4.1.1.1 | Character Constants | 4-4 |
| 4.4.1.1.2 | Character Strings | 4-4 |
| 4.4.1.1.3 | Character-to-integer Conversion | 4-4 |
| 4.4.1.1.4 | Integers | 4-5 |
| 4.4.1.1.5 | Pointers | 4-5 |
| 4.4.1.1.5.1 | Conversions | 4-5 |
| 4.4.1.1.5.2 | Relational Comparision | 4-5 |
| 4.4.1.1.6 | Register Variables | 4-5 |
| 4.4.1.1.7 | Unions | 4-6 |
| 4.4.1.1.8 | Globals | 4-6 |

4.4.1.2 Bitwise Logical Operations....... 4-6
4.4.1.2.1 Shifting....................... 4-6
4.4.1.2.2 Masking........................ 4-6
4.4.1.3 Coding Practices................. 4-6
4.4.1.3.1 Typedef........................ 4-6
4.4.1.3.2 Storage Alignment.............. 4-7
4.4.1.3.3 Variable Number of Function Arguments...................... 4-7
4.4.1.3.4 Bad Coding..................... 4-8
4.4.1.3.4.1 Order of Evaluation of Function Arguments........... 4-8
4.4.1.3.4.2 Order of Evaluation of Operator Expressions......... 4-8
4.4.1.3.4.3 Order of Assignment to Words and Integers, Byte Ordering.. 4-8
4.4.1.3.4.4 Embedded Machine Dependent Code........................ 4-8
4.4.1.3.4.5 Functions vs. Macros......... 4-8
4.4.1.4 Summary.......................... 4-9
4.4.2 Use of stdtyp.h.................... 4-9
4.4.3 Lexical Rules for Operators...... 4-10
4.4.4 Lexical Rules for Control Structures...................... 4-10
4.4.5 Lexical Rules for Functions...... 4-11
4.4.6 Preprocessor Symbols............. 4-12
4.4.7 References....................... 4-12

SECTION 5.0 LANGUAGE IMPLEMENTATION IDIOSYNCRACIES. 5-1

SECTION 6.0 PROGRAM IMPLEMENTATION CONVENTIONS .... 6-1
6.1 Use of Programming Templates ........ 6-1
6.2 COBOL Template ...................... 6-1
6.3 FORTRAN Template .................... 6-1
6.4 "C" Template ........................ 6-1

SECTION 7.0 IISS ERROR HANDLING PHILOSOPHY......... 7-1
7.1 Introduction ........................ 7-1
7.2 Overall Philosophy .................. 7-1
7.2.1 Error Definition................... 7-1
7.2.2 Error Handling Objectives.......... 7-2
7.2.3 Error Handling Philosophy.......... 7-2
7.2.4 IISS Subsystem Error Status Codes.. 7-3
7.2.5 IISS System Wide Error Status Codes 7-5
7.2.6 Conditional Data Items............. 7-8
7.2.7 Error Processing/Logging........... 7-8
7.2.8 Error Log File.................... 7-14
7.2.9 Error Messages.................... 7-14
7.2.10 Error Communication.............. 7-14
7.2.11 Utilities........................ 7-17
7.3 Network Transaction Manager ......... 7- 17
7.3.1 Overview.......................... 7-17
7.3.2 Error Codes....................... 7-18
7.3.3 Error Communication............... 7-18
7.3.4 Asynchronous Status Messages....... 7-18

7.4 User Interface ...................... 7-19
7.4.1 Overview......................... 7-19
7.4.2 Asynchronous Status Messages....... 7-19
7.4.3 Test-mode and Debug-mode........... 7-19
7.4.4 Error Communication................ 7-20
7.4.5 Message Management................. 7-20
7.5 Common Data Model Processor ......... 7-20
7.5.1 Precompiler........................ 7-20
7.5.2 Query Processing................... 7-20
7.6 Communications Subsystem ............ 7-21
7.7 IPC and IHC ......................... 7-21

## LIST OF ILLUSTRATIONS

| Figure | Title | Page |
|---|---|---|
| 6-1 | COBOL Program Template | 6-2 |
| 6-2 | Fortran Template | 6-4 |
| 6-3 | "C" Template | 6-5 |
| 7-1 | Standardized Error Codes | 7-6 |
| 7-2 | ERRPRO Module Specification | 7-10 |
| 7-3a | How to call ERRPRO from Cobol | 7-11 |
| 7-3b | How to call ERRPRO from "C" | 7-12 |
| 7-4 | SIGERR Definition | 7-15 |
| 7-5 | How to call SIGERR | 7-16 |

## SECTION 1

## INTRODUCTION

The purpose of this document is to identify and briefly explain guidelines and conventions to which the entire coalition should adhere throughout the various phases of IISS software development.

Several companies, each with different opinions relative to the content of the deliverable items for a development effort, are participating in the IISS coalition. It is important, therefore, that the coalition agree to specific development guidelines and conventions which will serve to focus efforts towards achieving a product characterized by maximal flexibility, understandability, and maintainability.

Guidelines and conventions proposed in this document are divided into three categories corresponding to major steps of the product life cycle: preliminary design, detailed design and program construction/testing. More specifically, the preliminary design corresponds to the first category in the Formulate and Justify Solution activity in the ICAM Life Cycle where, in preliminary design, system specification (SS), system design specification (SDS), development specifications (DS) et al. must be produced. The detailed design corresponds to the second category of the same activity wherein product specifications (PS) and unit test plans (UTP) must be produced. The third category contains program construction implementation testing which corresponds to the Construct and Integrate Solution activity of the ICAM Life Cycle wherein computer programs are actually installed and made ready for the user.

## SECTION 2

## PRELIMINARY DESIGN GUIDELINES/CONVENTIONS

### 2.1 Functional Requirements

IDEF0 will be used as the primary vehicle for conveying an understanding of IISS functional requirements. IDEF0 models will consist of appropriate function diagrams and associated text and glossary. Criteria for evaluating the quality and understandability of IDEF0 models will include the design goals of maximizing cohesion and minimizing coupling. Familiarity with these concepts can be gained by referring to any one of several Yourdon Press texts (e.g., "Structured Design" by Yourdon and Constantine, "Practical Guide to Structured Systems Design" by Page-Jones) or the latest IDEF0 manual.

### 2.2 Information Requirements

IDEF1 will be used as the primary vehicle for conveying an understanding of IISS information requirements. IDEF1 models will be evaluated in terms of well-known normalization rules (e.g., no repeat rule, non-null rule, home entity class for each attribute class).

### 2.3 IDEF1 Traceability

IDEF0 is the scoping tool for IDEF1 development. Therefore, entity classes and attribute classes included in IDEF1 models should be justified by and traceable to at least one IDEF0 ICOM. No formal technique will be used to correlate entity classes and attribute classes to ICOM's.

# SECTION 3

# DETAILED DESIGN

## 3.1 Module Specification

Each module which appears on a structure chart should be associated with a specification sheet which contains the following information:

- Identification - The module name in the form of a verb phrase. An abbreviation of the module name, limited to 7 characters, will also be identified and used as the actual program name of the module, if it is contemplated that the module will actually be implemented as a discretely and independently compiled program.

- Description - A brief description of the function performed by the module, including any special design or coding considerations. For lowest level modules pseudocode can be used if necessary and should follow the standards outlined in Section 3.2.

- Interfaces

  Entry Conditions - Descriptions (variable names, types, dimensions) for all input parameters. Description of global variables and acceptable ranges at the time of entry to the module.

  Exit Conditions - Descriptions (variable names, types, dimensions) for all output parameters, and the names of all modified global variables.

  Global Blocks - Name of global block and statement of its purpose. Description (variable name, type, dimension, initial values) for each variable contained in the block.

- Data Organization

  Local Variables - Descriptions (variable names, types, dimensions, initial values) for all significant local variables used by the module.

  Database Interaction - Names of all database entities and attributes used by this module. Description of the processing performed on these data items.

- Limitations - Restrictions and limitations of the code, including user, operator and programmer impact. Identification of tables and items which might require modification due to unexpected volume increases or overflows. Errors detected, error codes returned and subsequent action for each.

## 3.2 Procedural Descriptions

If necessary, pseudocode may be utilized for describing the procedural content of lowest level modules identified on structure charts. Guidelines and conventions relative to the development of pseudocodes specifications are described as follows:

### 3.2.1 Primary Purpose

The primary purpose of pseudocode is to enable the designer to express the control flow of a function in a straightforward, easy-to-understand manner, using a mixture of a normal English prose, selected verbs, and limited formal language expressions.

### 3.2.2 Level of Detail

Pseudocode should be written at a level of detail which permits direct coding of the program from it.

### 3.2.3 Basis for Development

Pseudocode is developed based on the processing portion of the structure chart, which specifies what functions are to be accomplished. The pseudocode specifications indicate when and how this processing will be performed.

### 3.2.4 Syntactical Rules

Three syntactical rules are always required:

1. Use capital letters to identify statements with special meanings, such as PERFORM UNTIL and END PERFORM.
2. Use indentation to identify statements to be performed within a loop or as otherwise necessary to promote clarity.
3. Indent with spaces, never use tabs. Interpretation of tabs is not consistent across all machines (e.g. IBM).

## 3.3 Suggestions for Complexity Adding

### 3.3.1 Module Number

Each module should call a maximum of 4 or 5 modules rather than 6 or more. Experience has shown that more than 6 modules may generate to much detail. A program which decomposes into 4 modules increases the likelihood that the actual code produced for that module will fit on one page.

### 3.3.2 Iterations

A module should contain at most 1 iteration. An alternative approach for nested iteration is to equate the innermost iteration(s) to a new module, thereby pushing complexity down to a lower level. This can lead to what Yourdon calls "procedural cohesion" which is not totally undesirable and can be justified for the sake of minimizing complexity.

Therefore, nested iterations are not desirable and are to be avoided.

## SECTION 4

## PROGRAM CONSTRUCTION GUIDELINE/CONVENTIONS

This section is divided into 2 subsections. The first subsection, 4.1, identifies and describes general guidelines and conventions thought to be independent of the characteristics of a specific implementation language. The second subsection, 4.2, presents guidelines and conventions germane to program construction using COBOL/Fortran/"C".

### 4.1 General Program Construction Guidelines/Conventions

#### 4.1.1 Design Documents Kept Current

All design documents should be kept current during coding.

#### 4.1.2 Program Banner

Each program will include a program banner which will identify the program and the configuration item of which it is a part. The banner immeditately preceeds a program header described in 4.1.3.

#### 4.1.3 Program Header

Each program will include a program header containing a subset of detailed design information. Specifically, the header will include:

Description
Entry Conditions
Exit Conditions
Global Blocks
Local Variables
Database Interaction

#### 4.1.4 Special Comments

The use of special comments should be limited. Except for unusual circumstances, the only form of special comment to be embedded within procedural sections of the program is a comment immediately preceding each call to a subroutine. For example, the following sequence of statements illustrates such commenting in a COBOL procedure division:

```
*
*         FIND OP PLAN SELECTED FOR DISPATCH
*
CALL FOPSFD USING STATUS
IF SUCCESSFULL
      *
      *
      *
```

### 4.1.5 Error Handling

Error handling will be inside the component (e.g. by a call to an error handling procedure) where the error is discovered and will follow a standard format, i.e., module name, line id, message, and debugging information. Error messages are output to a log file so that the information can be retained after development to facilitate system maintenance.

### 4.1.6 Message Storage

All messages issued by the system should be stored and accessed through tables. This facilitates message maintenance and error traceback.

### 4.1.7 Tables

Do not code tables into source programs when the tables may change with significant frequence. Such tables must be data sets which are read and loaded each time the program is executed.

### 4.1.8 Indentations

All indentations should be four spaces.

### 4.1.9 Nested IF Statements

When nested IF statements are required, indent them for clarity.

## 4.2 COBOL Language Guidelines/Conventions

The following are COBOL guidelines and conventions:

1. Represent subscripts, counters, etc. in binary (COMP-1).

2. Use the VALUE clause whenever possible to initialize WORKING-STORAGE instead of a MOVE statement.

3. Sequence COBOL programs as follows:

   o File Control - Show print files last.
     - Input files.
     - I/O files and sort files.
     - Output files.

   o File Section - Arrange FD in same order as SELECT statements in FILE-CONTROL section. Include record layouts in this section following applicable file descriptions.

   o Working Storage - Group data elements of a like kind, e.g.:
     - Switches like TRAN-EOF-SW and VALID-TRAN-SW.
     - Flags like STATUS-FLAG.
     - Control fields (areas reserved for data being sequenced).

- Print fields like LINE-COUNT and LINES-ON-PAGE.
- Counters like TRAN-COUNT and DUPLICATE-COUNT.
- Subscripts like RATE-TABLE-SUB.
- Tables.
- Report headings.

4. Increase all level changes in the record description by a minimum of two and always use two positions, e.g., 01,03,05,07.

5. Condition name, Level 88, entries must follow an elementary item and be indented four columns.

6. Condition names should be used to describe and test codes.

7. Statements are not to be placed on the same line as a paragraph name.

8. Each paragraph will be preceded by one or more blank lines (asterisk in Column 7).

9. Only one statement will appear on a line.

10. When using conditional statements, align the parts so that related IF and ELSE clauses are aligned.

11. All codes should adhere to the most current ANSI standard.

## 4.3 FORTRAN Language Guidelines/Conventions

The following are FORTRAN guidelines and conventions:

1. All codes must adhere to the most current ANSI standard. No nonstandard features shall be used, e.g. the names of all variables shall not exceed 6 characters.

2. All global data items will be defined as named common and defined as include files.

3. All variables shall be declared explicitly. All integer variables shall be explicitly defined as I*2 or I*4. This is to avoid confusion in relying on vendor default declarations which vary from I*2 to I*4. This is especially important for data that are passed as arguments.

4. Label numbers shall start at 10 and be incremented by 10.

5. Labelled statements shall not contain executable statements; CONTINUE statement shall be used.

6. Continuation lines shall be made by placing a '+' in column 6.

7. Comment statements shall be preceded and followed by empty comment statements, e.g.:

```
C**
C** comment
C**
```

## 4.4 C Language Guidelines/Conventions

### 4.4.1 Portability Issues

#### 4.4.1.1 Data Types

The use of stdtyp.h header file defined in a latter section of this document should eliminate most of the portability problems outlined in this section.

##### 4.4.1.1.1 Character Constants

The value of a character constant is the numeric value of the character in the machine's character set. For example, the character '0', zero, in ASCII is numeric 48, and in EBCDIC it is 240. To make a program independent of a particular numeric value, character constants should be written as a single character written within single quotes, such as '0'.

##### 4.4.1.1.2 Character Strings

A character string is a sequence of zero or more characters surrounded by double quotes. To the C compiler, a character string is an array whose elements are single characters. The compiler places the null character '\0' at the end of each character string so that programs can easily find the end. On some machines, there is a fixed limit to the length of a character string. For example, the Vax-11 C compiler has a limit of 1000.

##### 4.4.1.1.3 Character-to-integer Conversion

Character conversion to integer may produce a negative integer. On some machines, a character which has a leftmost bit as 1 will be converted to a negative integer because of sign extension. While on other machines, a character is promoted to an integer by adding zeros at the left end, and, thus, it will always be positive. For a machine's standard character set, the language definition of C provides that these characters will never be negative.

The getchar () function must be able to return to a variable all possible characters so it may be used for any type of input; therefore, its value must not be stored as a character but if stored should be stored into an integer variable. In addition, the special End of File (EOF) character may have the value -1 so that a comparison of a character variable to the EOF value on a machine which does not do sign extension will always fail.

#### 4.4.1.1.4 Integers

Positive integers are truncated toward 0 when divided, but if either of the operands is a negative integer, the method of truncation is machine dependent.

#### 4.4.1.1.5 Pointers

##### 4.4.1.1.5.1 Conversions

On many machines a pointer may be assigned to an integer and back again without changing it, no scaling or conversion takes place, and no bits are lost. Also, pointer variables of one type may be assigned to pointer variables of a differing type, i.e., a pointer which points to an integer value may be assigned to a pointer which points to a complex structure. But both types of a pointer assignment are nonportable since on some machines these assignments produce pointers which cause addressing exceptions when used.

Although pointers to one type may not be assigned to pointers to another type, an explicit type-conversion operator is frequently used to convert a pointer to one type to a pointer to another type. This operation is called casting. The following example illustrates type casting:

If p is declared: char *p, then (struct treenode *) p converts p from a pointer to an integer to a pointer to the specified structure.

Functions which return pointer values and arguments which hold pointer values should be declared as such even though the default type for functions and arguments is int. Not declaring pointer variables as such in not good practice since it relies on the details of implementation and machine architecture which may not hold depending on the particular compiler being used.

##### 4.4.1.1.5.2 Relational Comparisions

Although a pointer can be compared to an integer, the result of such a comparison is machine dependent except for comparison to the integer constant 0. A pointer variable containing a 0 is guaranteed not to point to any object and is considered a null pointer.

Pointer comparison between pointers when pointers point to objects in the same array is portable.

#### 4.4.1.1.6 Register Variables

Register variables are used to inform the compiler that these variables will be used heavily. Register variables are placed in machine registers. This capability may result in faster and smaller programs. Only automatic variables and formal parameters of a function may be declared as register. There are, however, some machine dependencies. The restrictions are placed on the types and number of effective register variables. For example, for some compilers, only the first three register declarations are effective, and their types may be selected from int, char, or pointer. Also for some compilers, pointer register variables to be effective, they be declared before other register variables. All excess or invalid register declarations are ignored.

#### 4.4.1.1.7 Unions

A variable which may contain objects of different types and sizes at different times is called a union. These types of variables are frequently used to provide a means of manipulating different kinds of data in one storage area but to allow the program to be free of any machine-dependent information. Unions, however, must be used carefully. If something is stored as one type and extracted as another type, the results are machine dependent. The program code must keep track of what type is currently stored in a union.

#### 4.4.1.1.8 Globals

Global variables must be defined exactly one time without the extern keyword and then may be used wherever needed with the extern keyword, i.e., they are defined once but may be referenced any number of times.

### 4.4.1.2 Bitwise Logical Operations

#### 4.4.1.2.1 Shifting

Shifting operations may provide different results on different machines. Right shifting, >>, a signed quantity on some machines will fill with sign bits (arithmetic shift); while on others, these bits will be filled as 0's (logical shift). A variable being used with such operations as shifting should be declared as unsigned since this declaration guarantees that the vacated bits will be zero-filled, not signed-bit filled, regardless of the machine the program is run on.

#### 4.4.1.2.2 Masking

Masking operations using the bitwise logical operator AND, &, should use mask constants which are independent of the word length. For example, to set the last 6 bits of x to zero:

```
x & ~077  is preferred to   x & 0177700
```

The second expression assumes that the variable x contains a 16-bit quantity, while the first expression is independent of word length.

### 4.4.1.3 Coding Practices

#### 4.4.1.3.1 Typedef

The C construct typedef provides a means for creating new data type names. This facility is useful for several reasons:

1. To parameterize a program against portability problems. If typedef's are used for the data types which may be machine dependent, then only these typedefs must be modified when porting the program.

2. A program is better documented when typedef's are used since a mnemonic name for a pointer to a structure, for example, LISTPTR, would be easier to understand than one declared as a pointer to a complex structure.

3. Future enhancements to the compiler or other utilities such as lint might make use of typedef declaration information to do some extra checking of programs.

### 4.4.1.3.2 Storage Alignment

Storage must be aligned properly for the types of object which are to be stored in it. Machines vary, but for each machine there is a most restricted type. The most restricted type means if this type can be stored at a specified address, then all other data types can be also. Normally, the compiler properly aligns the variables, but when a program does its own storage allocation of a data structure, a union should be declared to achieve proper alignment. This union should consist of the structure and the most restricted data type. For example:

```
typedef double ALIGNED; /* forces alignment            */
             /*    IBM 360/370    */

union treenode { /* tree node element */
      struct  {
              union treenode *leftptr; /* left branch*/
              int            nodeval;
              union treenode *rightptr; /* right branch*/
      } t;
ALIGNED dummy; /* force alignment when nodes */
               /* are allocated              */
};
typedef union treenode TREENODE;
```

The use of typedef and union handles alignment. The use of malloc/calloc for storage allocation eliminate the need for program-handled storage alignment.

### 4.4.1.3.3 Variable Number of Function Arguments

Functions which accept a variable number of arguments are not really portable because there does not exist a portable method of permitting the called function to determine the number of arguments actually passed in a given call. For example, the printf function is non-portable and must be changed for different environments if its to be able to handle a mismatch between the number of arguments and the format. If the function has only arguments of known types, it is possible, however, to mark the end of the argument list in a standard way, for example, using a special argument value that flags the end of the list.

4.4.1.3.4 Bad Coding

Some portability problems arise when questionable coding practices are used.

4.4.1.3.4.1 Order of Evaluation of Function Arguments

The order of evaluation of function arguments should not depend on in program code since the C language does not specify any order. On some machines, it is right to left but left to right on others.

4.4.1.3.4.2 Order of Evaluation of Operator Expressions

The language C does not define the order of evaluation of expressions. Most compilers will compute the expressions in the order that they deem is the most efficient, regardless of any side-effects caused by the expressions. Side-effects are the resulting change of some variable as a by-product of the evaluation of an expression. For example:

```
a[i] = i++
```

may evaluate differently depending on the order the compiler uses to evaluate this statement. The subscript could be the old value of the variable i or the new value. To insure a particular sequence of evaluations, temporary variables should be used.

4.4.1.3.4.3 Order of Assignment to Words and Integers, Byte Ordering

The order in which fields are assigned to words and characters are assigned to integers varies, right-to-left on some, left-to-right on others. A field is defined as a set of adjacent bits within a single int. A field is used to pack several objects into a single machine word. Frequently, this construct is used to create a set of single-bit flags in applications that need to use externally-imposed data formats such as hardware device interfaces that require the facility of addressing pieces of a word.

4.4.1.3.4.4 Embedded Machine Dependent Code

Programs that need to use information whose structure and content is maintained by machine's operating system should not embed actual declarations for this information within the programs themselves. This information should only be represented in standard header files which may be included in the programs.

4.4.1.3.4.5 Functions vs. Macros

Some functions may be implemented using the C-preprocessor macro facility. For example, an operation to find the maximum of two arguments could be defined using a macro. This definition does not need to take into account different data types as would a function written to do the same operation. As long as the arguments are treated consistently, the macro definition will work for any data type. The following illustrates such a definition:

```
     #define max(X, Y) ((X) > (Y) ? (X) : (B))
```

This macro will expand into in-line code whenever it is used.

### 4.4.1.4 Summary

This outline on portability issues is by no means exhaustive, but does provide an insight to the kinds of portability considerations that C programmers face when porting programs to different operating systems and machines. Purely hardware issues like word size and size of various data types can be addressed by using preprocessor macro definitions to test the type of machine or compiler the code is being developed for and compiled on. An appropriate synonym for the data type should be defined by using typedef's. This method is outlined in detail in the listing of the include header file stdtyp.h.

### 4.4.2.0 Use of stdtyp.h

Every C routine should include <stdtyp.h> first thing. Functions should then use only the following elementary data types which are defined in stdtyp:

| | |
|---|---|
| float | - single precision floating point |
| double | - double precision floating point |
| long | - 32 bit (or larger) signed integer |
| lbits | - 32 bits (or more) for bit manipulation |
| int | - natural size signed integer |
| unsigned | - natural size unsigned integer |
| bool | - natural size logical (zero / non-zero only) |
| short | - 16 bit (or larger) signed integer |
| ushort | - 16 bit (or larger) unsigned integer |
| bits | - 16 bit (or more) for bit manipulation |
| char | - single machine character |
| tiny | - 8 bit (or larger) signed integer |
| utiny | - 8 bit (or larger) unsigned integer |
| tbits | - 8 bits (or more) for bit manipulation |
| tbool | - 8 bit (or larger) logical (zero / non-zero only) |
| metachar | - 16 bit (or larger) augmented character (signed) |
| void | - function that returns no value |
| fortran | - storage class for foreign (non-c) routines or C routines which are callable from foreign routines |

The natural size items should be used for maximum efficiency if at least 16 bits is sufficient and you don't really care about the actual length. Tiny numbers should only be used where space is critical since they may involve significantly more processing than other sizes.

Since not all compilers support ushort, tiny, and utiny, the functions USHORT(), TINY(), and UTINY() should be used whenever referencing them (e.g. a = TINY(x);).

The following utility macros are also defined in stdtyp and should be used when appropriate:

| | |
|---|---|
| LURSHIFT(n, b) | - unsigned long right shift of b for n bits |
| MAX(a, b) | - maximum of a and b |
| MIN(a, b) | - minimum of a and b |
| ABS(a) | - absolute value of a |
| STRASN(a, b) | - portable a = b for structures |
| NULL | - null pointer value |
| TRUE | - 1 FALSE - 0 |
| SUCCESS | - Success status for exit |
| FAILURE | - Failure status for exit |

## 4.4.3.0 Lexical Rules for Operators

Unary operators should be written with no space between them and their operands (EXCEPTION: sizeof should be followed by a blank).

The operators "->", "." and "[]" should also have no space around them.

Assignment operators and the conditional operator should always have space on both sides.

Other binary operators should usually have space around them, but may appear with no space if they are of higher precedence than other operators in the same expression.

Keywords are always followed by one blank.

There should be no space between a function name and its argument list.
Examples:

```
x = -a * b;

for (i - 1; i < n; i++) printf("%s%c", str[i], (i < n-1) ? ' ' :
'\n');

s = a*a + b*b;
```

## 4.4.4.0 Lexical Rules for Control Structures

Every line which is part of the body of a C control structure is indented one tab stop farther than its controlling line unless the body is a single statement, in which case it may appear immediately following the control structure on the same line.

If braces are used, each opening or closing brace appears on a line by itself, indented to the same level as the enclosed statements.

For multiple choice constructs where "switch" is not appropriate, the "else if" construct should be used. Each "else if" (and trailing "else", if used) is indented to the same level as the initial "if".

Braces should always be used for "do - while" with the "while" part following the closing brace on the same line. This eliminates possible confusion with "while".

Examples:

```
if (x[i] >) max = i;

if (a == 1)
   {
   init (x);
   a = 2;
   }

if (a > b)
   if (a > c) max = a;
   else max = c;
   else if (b > c) max = b;
   else max = c;

if (a.local)
   {
   ...
   } else if (a.global)
   {
   ...
   } else
   {
   ...
   }

do {
   ptr = ptr->next;
   } while (ptr->node != inode);
```

## 4.4.5.0 Lexical Rules for Functions

The order of statements in a source file should be: preprocessor #include statements, preprocessor #define statements, external variable definitions, and functions, all beginning at the left margin.

Within a function, variable declarations should be separated from executable code by a blank line.

Example:

```
#include <stdtyp.h>
#include <stdio.h>
```

```
#define BUFSIZ 100

FILE *infile;

char *fillbuf(buf) /* fill buffer */
   char buf[BUFSIZ];
   {
   int i;
   char *ptr = buf;

   for (i = o; i < BUFSIZ; i++) *ptr++ = getc(infile);
   return (buf);
   }
```

## 4.4.6.0 Preprocessor Symbols

The following preprocessor symbols are available for handling system dependent code:

| | |
|---|---|
| ebcdic | - defined if the system character set is EBCDIC instead of ASCII |
| vax11c | - defined for vax-11 c compiler |
| whitesmith | - defined for whitesmith c compiler |
| ibm | - defined for Bell C/370 compiler |
| iswb | - defined for IS/Workbench compiler |
| vms | - defined when the operating system is VMS |
| os370 | - defined when the operating system is OS/370 |
| unix | - defined when the operating system is unix |

These symbols can be used to escape to a standard operating system routine instead of a C routine. (For example, a bit manipulation routine could call the VMS run-time library routines that provide access to the VAX bit manipulation instructions if vms is defined, but use the C bit manipulation operators otherwise.)

## 4.4.7.0 References

Plum, Thomas, C Programming Standards and Guidelines - Version U (Unix and Offspring) (3rd edition, Plum Hall Inc, Cardiff, NJ, January 1982.

## SECTION 5

## LANGUAGE IMPLEMENTATION IDIOSYNCRASIES

To ensure maximum portability of IISS software, not only should ANSI COBOL 74 and Fortran 77 standards be adhered to, but also it is necessary to recognize the difference in implementation mechanisms of the standards by different computer vendors. These differences are attributed to variations in computer architecture. The most significant differences lie in data representation and program organization. As these discrepancies are identified, information relating to them should be distributed among the IISS team. Until a complete well-documented list has been created, developers should avoid using as many vendor-dependent features as possible. Currently, the following machine-related differences have been identified:

1. The Vax Cobol compiler will initialize data items depending on the PICTURE and VALUE clauses. If there is no VALUE clause, the VAX compiler will initialize the field to zeroes if numeric, UNDEFINED if indexed data item or INDEX, and SPACES for anything else. The IBM Cobol compiler will only initialize values if the VALUE clause is specified; if not specified, the everything is UNDEFINED. The recommendation is made that all variables have VALUE specified for initialization purposes.

2. Casual conversion betweens PIC 9, PIC X, and various COMPUTATIONAL variables must be carefully controlled as different interpretations of the information as far as size and internal representation may occur on different machines.

3. Quotation marks and apostrophes, as well as underbars and dashes, are being randomly tossed about in Cobol and Fortran programs to the point that entire program sets may not compile on another machine without correcting for conversion. This must be carefully checked for each language in order to insure portability.

## SECTION 6

## PROGRAM IMPLEMENTATION CONVENTIONS

### 6.1 Use of Programming Templates

Programming templates are introduced into program modules when they are first being written from scratch. The templates provide a common header format so that all programs can be quickly understood by anyone picking up the module for the first time. The particular formats must also be adhered to so that our Fully Automated Documentation System can pick up relevant information it needs. This then provides a variety of information about modules and their interactions with each other without having to read the actual code which was written. The templates form the common convention to which all programs must adhere.

### 6.2 COBOL Template

The COBOL program template is depicted in Figure 6-1. If any comment on the template does not apply, the programmer can choose to put in (NONE) or delete the comment from the format. For example, if a program does not have any include files, the comment for INCLUDE FILES can be deleted from the program or the programmer can insert an additional comment line to indicate (NONE).

### 6.3 Fortran Template

The FORTRAN program template is depicted in Figure 6-2.

### 6.4 "C" Template

The "C" program template is depicted in Figure 6-3.

```
IDENTIFICATION DIVISION.
       PROGRAM-ID. ??????.
      *
      **********************************************************
      *                                                        *
      *   PURPOSE: A 1-LINE SYNOPSIS OF WHAT IT DOES AND WHY   *
      *                                                        *
      **********************************************************
      *
      * DESCRIPTION :- ????? (a more detailed description)
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.  ??????.
       OBJECT-COMPUTER.  ??????.
      *
       DATA DIVISION.
      *
      * DATA ORAGANIZATION :-
      *
       WORKING-STORAGE SECTION.
      *
      * INCLUDE FILES
      *
       COPY ????????.
      *
      * LOCAL VARIABLES
      *
       ?
      *
      * INTERFACES :-
      *
       LINKAGE SECTION.
      *
      * GLOBAL DATA ITEMS
      *
       ?
      *
      * INPUT ARGUMENTS
      *
       ?
```

Figure 6-1. COBOL Program Template

```
*
* OUTPUT ARGUMENTS
*
 ?
*
* LIMITATIONS :-
*
 ?
*
* PROCESS DESCRIPTION :-   (optional)
*
 PROCEDURE DIVISION.
*
 START-PROGRAM.
*
 ?
*
* RETURN
*
 EXIT-PROGRAM.
*
     EXIT PROGRAM.
 ?
*
* PROCESS ERROR
*
 COPY ERRPRO.
```

Figure 6-1(continued). COBOL Program Template

```
C(
      SUBROUTINE XXXXXX (ARG1, ARG2)
C**
C**
C**********************************************************
C**                                                      **
C**   PURPOSE: A 1-LINE SYNOPSIS OF MODULE            **
C**                                                      **
C**********************************************************
C**
C** DESCRIPTION :-
C**
C** ??????? (a more detailed description; multiple lines)
C**
C**
C** INTERFACES :-
C**
C** INPUT ARGUMENTS
C**
?
C**
C** OUTPUT ARGUMENTS
C**
?
C**
C** INCLUDE FILES
C**
?
C**
C** GLOBAL DATA ITEMS
C**
?
C**
C** DATA ORGANIZATION :-
C**
C**
C** LOCAL VARIABLES
C**
?
C**
C** LIMITATIONS :-
C**
?
C)
C**
C**
C** PROCESS DESCRIPTIONS
C**
?
      END
```

Figure 6-2. Fortran Template

```
/*--------------------------------------------------------------
 *
 * NAME: main
 *
 * PURPOSE: TO CREATE A .DOC FILE TO FIT FADS SPECIFICATIONS
 *
 * SYNOPSIS
 *    MODULE NAME(arg1,arg2...)
 *
 *
 * INPUTS/OUTPUTS
 *
 * INPUTS:
 *           arg1
 *           ????
 *
 * OUTPUTS:
 *           arg2
 *           ????
 *
 * DESCRIPTION:
 *            A COMPLETE DESCRIPTION AS YOU SEE FIT.
 *
 */
? (rest of program)
```

Figure 6-3. "C" Template

# SECTION 7

# IISS ERROR HANDLING PHILOSOPHY

## 7.1 Introduction

This plan presents an overall philosophy in error handling for IISS. The philosophy emphasizes the generalities and not the specifics for the handling of a particular error by a subsystem. The overall philosophy for all subsystems will be presented first. Then the procedures applying to each subsystem will be discussed individually. Where the individual subsystems do not currently adhere to this IISS Error Handling Philosophy, changes will be made in a gradual manner as new versions of modules are produced.

## 7.2 Overall Philosophy

### 7.2.1 Error Definition

Errors that can be encountered in IISS can be classified into one of four catagories. These catagories are defined as:

1. NON-FATAL INFORMATIONAL ERRORS: The User needs to get information returned to him. An AP has noticed that the user has done something wrong, but is still up and running. It is up to the AP to return the needed information to the user through some screen-display function. The error-handling routines discussed here will not handle this type of error because, basically, it is not an error. It is mentioned here only for clarity.

2. FATAL ERRORS: Errors which an AP does not expect during normal execution but the IISS system can act upon and relay back to the AP and the user (e.g.: bad database calls, mailbox-full timeouts, and any other errors which are recoverable but will probably cause the user's AP to fail. This type of error will be logged to the error log file because it shows errors whose recovery procedures were not programmed in and could cause an AP to abort.

3. CATASTROPHIC ERRORS: Errors that don't cause an entire IISS sub-system process to abort yet cannot be resolved by the IISS program (e.g.: channel cannot be assigned, bad initiation). This type of error must be logged and must be "bubbled-back" to higher modules and to the originating AP, if there is one. The User will also be informed of the impending disaster.

4. IMPOSSIBLE ERRORS: Errors which cause an IISS process to abort (e.g.: invalid decimal data, divide by zero). These errors should be captured using the operating system's trap handler and should be logged to the error log file. (Note: The trap handler facility has not been implemented in IISS.) We would hope that these get

logged, but the IISS system may get caught in a loop, for example, and would therefore never return to log it (i.e.: it crashed).

### 7.2.2 Error Handling Objectives

The overall objectives of error handling are:

1. ALL error conditions should be uniquely and concisely defined.

2. All error codes should be stored in files which are maintained by the UI's Message Management (MM) facility. No error code should be hard coded into source code.

3. Errors should be logged so as to help software developers locate problems.

4. Errors should be reported through the UI so as to help users to determine the status of a request.

5. Error status should be returned so as to allow other subsystem interfaces to recover from the problem if possible.

6. Errors should be logged so that IISS users who are not intimately familiar with IISS subsystem code can understand and possibly narrow down the cause of an error.

### 7.2.3 Error Handling Philosophy

When detecting an error, a module should process the error using the following alternatives:

1. Based on the error, the module must return an IISS status code to the calling module. After receiving an error status code from a call to other subsystem services such as NTM, IPC, the operating system, or ORACLE, the error status code should be logged to the error log file without any translation if it is fatal. More information can be added to the log using the message description. For modules that issue operating system macro calls, it is good practice to log all system error messages in the log file to facilitate debugging.

2. Upon return from a lower level module or other subsystem services, the higher level module should always check the returned status code to determine whether an error has occurred. An errorless run should never be assumed if the call returns an error status code. If an error has occurred, the higher level module can choose to do one of three things:

A. Recover from error and continue execution.

B. Invoke the error processing routine and then return control to the calling module while passing the error status back.

C. Call SIGERR which:
a) signals the user with a message about the error that occurred
b) invokes the error processing routine and then the AP may exit.

3. When a catastrophic error occurs, the software should try to contain the error without affecting the operation of other system components. In the worst case, NTM should always abort the application processes to ensure its own survival.

4. Fatal errors should be logged using the "bubble-back" technique where the error is progressively passed back up the calling chain. The error is put in the log at each calling level with a message pertinent to that level. In the following example VALMBE is the lowest level module and COMM02 is the highest:

```
- 84/03/12  15:34:46  VANDERMINDEN  VALMBE  10015
   WHILE VALIDATING MAILBOX ALREADY EXIST
- 84/03/12  15:34:46  VANDERMINDEN  CRTMBX  10015
  WHILE CHECKING EXISTENCE OF MAILBOX
- 84/03/12  15:34:46  VANDERMINDEN  INICX2  10015
  WHILE CREATING 'COMMIN2' MAILBOX
- 84/03/12  15:34:46  VANDERMINDEN  INTCM2  10015
  WHILE ESTABLISHING MAILBOX CONNECTION
- 84/03/12  15:34:46  VANDERMINDEN  COMM02  10015
  DURING INITIALIZATION
```

5. If an impossible error is detected, the operating system's trap handler should be invoked and, if possible, the error should be mapped to a fatal IISS status code.

6. Each AP whose execution can be initiated from an IISS terminal should inform the user or the operator of the status of a request upon failure. This applies to the CDM precompiler as well as any other IISS system AP's such as UIMS applications. A call to SIGERR (as opposed to just ERRPRO) would be appropriate to accomplish that.

### 7.2.4 IISS Subsystem Error Status Codes

Each subsystem will create and maintain one or more error status code file(s) using the message management facility. The files will contain all error status codes that may affect the subsystem. These error codes will be of an X(5) format. The error codes for each subsystem will begin with a different digit. This allows for easy identification and avoids the overlapping of error codes between subsystems. The error code prefix is assigned as follows:

1- IPC

2- Communications

3- NTM

4- CDMP

5- Application Interfaces

6- IBM Interface

7- User Interface

8,9- reserved for future subsystems

As more subsystems are added to IISS, this list will be updated.

The second and third digit from the left of the error status code may be used by each subsystem to identify the subsystem specific category of the error.

An error code of zero represents the successful or OK condition.

Since the error status codes are of an X(5) format, the characters can be alphabetic. This opens up the number of possible error status codes and prefixes for future IISS applications. This also allows the error code prefixes to correspond to the application subsystem designations. In order to use alphabetics, though, the Message Management Utility of the User Interface may have to be modified.

The naming conventions for the error status files have the following prefix for file names:

1. ERRCOx--COMM
2. ERRNTx--NTM
3. ERRCDx--CDMP
4. ERRUIx--UI/VTI
5. ERRIPx--IPC
6. ERRSYx--System Wide Errors

The 'x' in the prefix is an alphanumeric character and is assigned by the subsystem.

When programming in COBOL, program variable names should use the prefix of 'KES' for all error status codes. The remaining characters in the variable name should convey the description of the error status code either in readable form or mnemonics. The correspondence between variable names and the unique error status codes are to be put in copy files. These copy files should also contain comments indicating what descriptions the mnemonic field names correspond to.

To insure portability in the other programming languages, (i.e.: FORTRAN and C), the first six characters in the variable name have to be unique. Because of this size limitation, the 'KES' prefix does not have to be used.

### 7.2.5 IISS System Wide Error Status Codes

System wide standardized AP status codes have been defined for some frequent error codes such as for file errors, DBMS errors, table errors, system service (SORT/MERGE) errors, or memory allocation errors. These codes will have program variable names which are unique in the first six characters. This is to ensure language-independence and will allow all source code to use the same program variable name for a particular error code.

The system wide standardized error codes will be maintained by the message management (MM) facility. The MM has a utility called INCGEN. This utility will take the error codes and their corresponding variable names and generate include files for the various IISS programming languages.

Figure 7-1 contains a list of these standardized error codes. This list will be expanded in the future as additional system-wide error status codes are identified.

| TYPE OF ERROR | ERROR CODE | ERROR DESCRIPTION |
|---|---|---|
| MEMORY ALLOCATION ERRORS | "00001" | NOT ENOUGH MEMORY AVAILABLE |
| DBMS ERRORS | "01000" | DATABASE ERROR |
| | "01001" | DATABASE ALREADY READIED |
| | "01002" | DATABASE NOT READIED |
| | "01003" | DATABASE CANNOT BE OPENED |
| | "01004" | DEADLOCK |
| | "01005" | NO DUPLICATES ALLOWED |
| | "01006" | END OF SET |
| | "01007" | RECORD NOT A MEMBER OF SET |
| | "01008" | RECORD ALREADY MEMBER OF SET |
| | "01009" | RECORD TYPE NOT CURRENT |
| | "01010" | ROW NOT FOUND IN TABLE |
| | "01011" | BAD DML STATEMENT |
| | "01012" | DATABASE ROLLBACK ERROR |
| | "01013" | DATABASE COMMIT ERROR |
| FILE ERRORS | "02000" | FILE OPEN ERROR |
| | "02001" | FILE NOT FOUND |
| | "02002" | FILE NOT CREATED |
| | "02003" | FILE NOT DELETED |
| | "02004" | END OF FILE |
| | "02005" | FILE NOT OPENED |
| | "02006" | FILE NOT CLOSED |
| | "02007" | READ ACCESS DENIED |
| | "02008" | WRITE ACCESS DENIED |
| | "02009" | FILE NOT INDEX SEQUENTIAL |
| | "02010" | FILE NOT SEQUENTIAL |

Figure 7-1. Standardized Error Codes

| TYPE OF ERROR | ERROR CODE | ERROR DESCRIPTION |
|---|---|---|
| TABLE ERRORS | "03000" | TABLE OVERFLOW |
| | "03001" | TABLE NOT FOUND |
| | "03002" | TABLE ENTRY NOT FOUND |
| | "03003" | TABLE FULL |
| | "03004" | TABLE WRITE ERROR |
| | "03005" | TABLE READ ERROR |
| SORT/MERGE ERRORS | "04000" | SORT PACKAGE ERROR |
| | "04001" | INVALID KEY SPECIFICATION |
| | "04002" | FILE SIZE INVALID |
| | "04003" | ERROR CLOSING INPUT FILE |
| | "04004" | ERROR CLOSING OUTPUT FILE |
| | "04005" | ERROR OPENING INPUT FILE |
| | "04006" | ERROR OPENING OUTPUT FILE |
| | "04007" | ERROR READING FILE |
| | "04008" | RAN OUT OF WORK SPACE |
| | "04009" | KEY SPECIFICATION MISSING |
| | "04010" | TOO MANY INPUT FILES |
| | "04011" | ERROR WRITING FILE |

Figure 7-1(continued). Standardized Error Codes

The following conventions should be followed in assigning additional codes:

1. First character of the error code is "0"
2. Second character corresponds to the type of system wide error as follows:

| | |
|---|---|
| 0- | Memory Allocation |
| 1- | DBMS Errors |
| 2- | File Errors |
| 3- | Table Errors |
| 4- | SORT/MERGE Errors |
| 5- | NDDL Errors |
| 6- | 9--Additional Types |

### 7.2.6 Conditional Data Items

For each error status file for each subsystem, there should be an equivalent check status file. The three character prefix for the file name is 'CHK' instead of 'ERR' for these files, but the last three characters should correspond to their counterpart error status files. This file will contain all conditional data items to help in checking for error conditions. The values of these conditional data items are equal to all the error status codes.

If programming in COBOL, each conditional data item is prefixed with 'QCS'. The remaining characters in the variable name should convey the description of the error status code either in readable form or mnemonics. The correspondence between variable names and the unique error status codes are to be put in copy files. These copy files should also contain comments indicating what descriptions the mnemonic variable names correspond to.

To insure portability in the other programming languages, (i.e.: FORTRAN and C), the first six characters in the variable name have to be unique. Because of this size limitation, the 'QCS' prefix does not have to be used.

### 7.2.7 Error Processing/Logging

There is a standard error processing routine written for each IISS host computer called ERRPRO. This routine is invoked to log all errors by any module using the following arguments:

1. IISS error code in the status field.
2. The six character name of the module where the error is detected

3. Necessary debugging information to help the programmer to locate the error including operating system dependent error codes (e.g. VAX system service error code or Level 6 macro status code). This message description should be used in describing the conditions under which the error occurred, and not for explaining the error code.

This common error processing routine will date and time stamp the error, format the error message, and write the error message to the error log file (ERRLOG.DAT). This routine does not terminate the calling program. It returns control to the calling program which will "bubble-back" the return status to the top level module. Only the top level module can terminate the program. Figure 7-2 contains a description of the Error Processing module (ERRPRO). Figures 7-3(a) and 7-3(b) describe the constructs for calling ERRPRO from a Cobol module and a C module, respectively.

IDENTIFICATION :

ERRPRO -- Process/Log Error

DESCRIPTION :

This module is used to process errors by logging them on a file. There will be a ERRPRO routine in each host computer. The module will make use of operating system characteristics to perform the error logging. This module will date and time stamp the error message and format the error message before logging the message on a file.

INTERFACES :

ENTRY CONDITIONS :

```
RET-STATUS PIC X(5).
MODULE-NAME PIC X(6).
MESG-DESC PIC X(60).
```

EXIT CONDITIONS :

(NONE)

GLOBAL BLOCKS :

(NONE)

DATA ORGANIZATION :

LOCAL VARIABLES :

(NONE)

DATABASE INTERACTION :

(NONE)

LIMITATIONS :

(NONE)

Figure 7-2. ERRPRO Module Specification

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CRTMBX.

DATA DIVISION.

WORKING-STORAGE.

01  MODULE-NAME                PIC X(6) VALUE 'CRTMBX'.
01  MESG-DESC                  PIC X(60).

LINKAGE SECTION.

01  RET-STATUS                 PIC X(5).

PROCEDURE DIVISION USING RET-STATUS.

        CALL "CRTMBX" USING mailbox-name,
                            RET-STATUS
        MOVE RET-STATUS TO CHECK-STATUS
        IF SUCCESSFUL
                .
                .
                .
        ELSE
            MOVE "WHILE CALLING CRTMBX " TO MESG-DESC
            PERFORM PROCESS-ERROR.

EXIT-PROGRAM.
     EXIT PROGRAM.

*
* PROCESS ERROR
*
 COPY ERRPRO OF IISSCLIB.  (This include file contains
the call to ERRPRO. The paragraph name is PROCESS-ERROR. The
following is a listing of the copy file).
                PROCESS-ERROR.
                    CALL 'ERRPRO' USING RET-STATUS,
                                        MODULE-NAME,
                                        MESG-DESC.
```

Figure 7-3(a). How to Call ERRPRO from COBOL

```
/* NAME
          *    EXMPLE -
          *       Written:  1-AUG-1984 12:00:00
          *       Revised:  1-AUG-1984 12:00:00
          *
          * SYNOPSIS
          *    EXMPLE()
          *
          *
          *    Inputs/outputs:
          *
          *    Inputs:
          *             error code
          *
          *    Outputs:
          *
          *
          * DESCRIPTION
          *      This is an example of how to call the
          *      COBOL routine, ERRPRO,
          *      from a "C" module.
          *      If an error occurs, get the error code,
          *      module name, set up
          *      the error message and pass them to ERRPRO.
          */

        #include <stdtyp.h>
        #include <stdio.h>
                           .
                           .
                           .
        char *exmple()
           {
                           .
                           .
                           .
           char   mesg_desc[60]; /* area used for error mess.*/
           char   *ret_status[5];   /* error code */

           char   *module_name = "EXMPLE";   /* module name */
           char   ntm_buf[BUFDIZE];
```

Figure 7-3(b). How to Call ERRPRO from C

```
        int            ntm_bufsiz;
                           .
                           .
                           .
                initex(ntm_buffer,
                       ntm_bufsiz,
                       ret_status);
                if
(strncmp(ret_status,GOOD_INITEX,RET_CODE_LEN) != 0)
                 {
                    fndmsg(ret_status,mesg_desc);
               /* find message string to fit error code */

                    errpro(ret_status,
                           module_name,
                           mesg_desc); /* pass error code,
                                    module name and message
                               description to will format and
                             send error message to error log */
                           .
                           .
                           .
                 }
                           .
                           .
                           .

            return(ret_status);
        }
```

Figure 7-3(b)(continued). How to Call ERRPRO from C

While executing ERRPRO if a problem occurs in writing to the error mailbox, the message is written to a file called ERRFTL.DAT.

### 7.2.8 Error Log File

There will be a different log file maintained on each host called ERRLOG.DAT. The standard error processing routine on each host will log error messages on its respective error log file. Each record in the error log file contains the following fields:

1. date and time of message logged
2. name of process logging the error
3. name of the module where the error is detected
4. IISS error code or operating system dependent error code
5. message description information

### 7.2.9 Error Messages

An error message file will be maintained on the VAX containing the defined meaning of all error codes in IISS. This file will be maintained by the UI message management services. A common error processing routine, if desired, can make use of this file to print out a user friendly message.

### 7.2.10 Error Communication

A service called SIGERR is available for AP'S to send error information to the IISS user and operator, as well as log information to ERRLOG.DAT through ERRPRO. The information will be sent to the User Interface when it is the original source of the AP which called SIGERR and when it is in "test-mode" or when it is an Informational Error. Figure 7-4 contains the definition of the SIGERR call. Figure 7-5 contains the construct for calling SIGERR from a Cobol module.

```
INDENTIFICATION :        SIGERR

DESCRIPTION : The service, SIGERR, is available for AP's to send
        error information  through the IISS.  The error
        information will be sent to the following:

    a) the IISS operator's console
    b) the UI AP when it is the original source of the AP
        who called SIGERR and when it is in "test-mode"
    c) ERRPRO, who will get the error logged to ERRLOG.DAT

        CALL DEFINITION :
              CALL "SIGERR" USING ERROR-CODE
                                  SEVERITY-LEVEL
                                  ERROR-MSG
                                  RET-CODE.

        INPUT ARGUMENTS :
                            ERROR-CODE PIC X(5).
                            SEVERITY-LEVEL PIC X.
                            VALUES:
                            W--WARNING;Non-standard use of
                                       feature which may cause
                                       trouble
                            I--INFORMATIONAL;Always sent to the
                                             UI and always
                                             displayed
                            E--ERROR;Not fatal but may cause
                                     improper execution
                            F--FATAL
                            ERROR-MSG PIC X(72).
                             (This field contains additional
                              information about the error.)
                             SPACE FILL IF NOT USED

        OUTPUT ARGUMENTS :
                            RET-CODE

        RET-CODE VALUES :
                            SIGERR-SUCCESSFUL
                            SIGERR-UNSUCCESSFUL (Indicates that
                                 the service was not successful
                                 in sending a message with the
                                 AP error information.)
```

Figure 7-4.  SIGERR Definition

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  CDCE1.

DATA DIVISION.

WORKING-STORAGE SECTION.
01  MODULE-NAME     PIC X(6)  VALUE "CDCE1".
01  MESG-DESC       PIC X(60).
01  RET-STATUS      PIC X(5).
77  SEVERITY-LEVEL            PIC X.
77  ERROR-MSG                  PIC X(72).
        .
        .
COPY SRVRET OF IISSCLIB.
        .
        .
PROCEDURE DIVISION.
START-PROGRAM.
        .
        .
    CALL "RCV" USING  NTM-LCHAN,
                     NTM-WTFLG,
                     NTM-SOURCE-DEST,
                     NTM-MTYP,
                     NTM-DLEN,
                     CS-ES-MESS-IN,
                     ACCEPT-STATUS,
                     NTM-MSNO.
    IF NOT RCV-NORMAL-MESSAGE
        MOVE "RCV CALL FAILED"  TO  MESG-DESC
        MOVE ACCEPT-STATUS      TO  RET-STATUS
        PERFORM PROCESS-ERROR
        MOVE "F" TO SEVERITY-LEVEL
        MOVE "RCV CALL FAILED" TO ERROR-MSG
        CALL "SIGERR"  USING  RET-STATUS
                              SEVERITY-LEVEL
                              ERROR-MSG
                              RET-CODE
        GO TO EXIT-PROGRAM.
             .
             .
EXIT-PROGRAM.
             .
             .
COPY ERRPRO OF IISSCLIB.
```

Figure 7-5.  How to Call SIGERR

The error message field is available for the programmer's use to include additional or variable information about the error. This field maps to the descriptor field in ERRPRO for messages that are destined for the user's terminal. This field might contain such information as the subroutine name or filename associated with the error. Its use is for helping the user by giving information that could be helpful in diagnosing and fixing an error or resource problem. This information will be output on the user's terminal with the error code when the user has "test-mode" set or if the error is designated as informational.

### 7.2.11 Utilities

On each IISS computer there will reside an error log file (ERRLOG.DAT). This file will contain all error messages. To enable the IISS' users and developers easy access to this file, an Error-Retrieval utility has been developed. This utility allows for a variety of different ways to access this file and extract the information in question. For example, if a user wants to look at errors between a certain time period, the utility can copy all messages for that time period either onto the CRT screen or into another data file that the user has specified.

When IISS gets to the point where it will communicate with two or more computers, the error logs from the other computers will periodically be moved to the VAX for storage and perusal. A utility will be written to merge the error logs by date/time. (This approach assumes that the clocks are close to the same.)

A mechanism is in place to periodically start a new error log file. This will keep the error log file to a manageable size.

The NTM subsystem has a file of its own in which it records all messages, including error messages (NTMLOG.DAT). A utility similar to the one developed for the error log file will be provided. These utilities will allow for a variety of ways to access and extract the information from this file.

## 7.3 Network Transaction Manager

### 7.3.1 Overview

The NTM's AP error handling can be categorized into local APC and remote APC handling. Basically, all local message processing errors (e.g. bad service arguments, authority violations, local APC resources unavailable) can be determined by the requesting AP in the return argument of the NTM Service Call. On the other hand, errors that occur in handling the destination AP of a request (e.g. entries not found in NTM tables, resources not available at the remote APC, errors in process starts) are handled with NTM "asynchronous" error messages that are sent to the NTM at the immediate source of the request. An AP will receive these messages only if its characteristics are appropriately set in the NTM AP Characteristics table. The "asynchronous" error messages that are available are those associated with the abnormal termination or start of the destination AP. The UI and CDMP can check for these "asynchronous" messages in order to establish the status of spawned AP's. The NTM uses the following means to handle its own errors:

1. display error messages on the operator console

2. log all messages to the NTM log (NTMLOG.DAT)

3. call ERRPRO to log error messages.

### 7.3.2 Error Codes

The NTM error codes will be defined to clearly identify the error conditions. There are several general types of errors (e.g. unsuccessful AP start, abnormal termination) that are indicated by the first three characters of the five character error code. Specific problems are indicated in the last two characters of the error code (e.g. missing executable). This leaves APs that can not act on the specific information, to take action on the general information of the first three characters; while APs such as the UI can use the specific information encoded in the last two bytes to inform the user of events that might be of interest to him (e.g. destination APC down). The information encoded in the error code is available to the IISS operator in the message log (NTMLOG.DAT) so that the exact problem can be determined for APs that do not decode the last two bytes.

### 7.3.3 Error Communication

The NTM supplies a service called SIGERR to send error information to the IISS user and operator. This service outputs a message containing the error code and variable error information on the operator's console. This error is also logged in the NTM's log file (NTMLOG.DAT). SIGERR also sends a message to the User Interface (UI) if the UI's "test-mode" is in effect or if the error is designated as informational. The UI will output the error code, error description, and variable error information on the IISS user's terminal if its "debug-mode" is in effect or the error is designated as informational. (Refer to section 7.4.3 for more information on test-mode and debug-mode.)

Two operator commands are available to enable (SE) or disable (SO) error message reporting on the operator's console. SO is the default condition. Setting SE will cause the error messages to be displayed in real time on the IISS operator's console.

### 7.3.4 Asynchronous Status Messages

The NTM sends messages regarding AP status to the UIMS so the UIMS can display the message on the IISS user's terminal. This message is of type 'SE' and contains error code, severity code, offending AP name, and error message. If the UI's "test-mode" and "debug-mode" are set, the NTM will send all messages that are displayed on the operator console to the UIMS so the UIMS can display them on the terminal. Informational status messages will always be sent to the UIMS and will always be displayed.

## 7.4 USER INTERFACE

### 7.4.1 Overview

The User Interface is the IISS interface to the IISS user's terminal. The UI uses ERRPRO to log all its errors to the error log file. To help the user in dectecting errors, the form processor (UI) allows for a debug mode where all error messages are displayed on the terminal.

On detecting an error within the UI due to NTM, IPC, or ORACLE calls, the UI should display a message on the terminal informing the user of the condition. This allows the user to determine whether the request is successful or not. If a fatal error is detected that may affect the future operation of the UI, the UI should pass a message to the NTM using SIGERR so that the NTM can display the message on the operator console.

### 7.4.2 Asynchronous Status Messages

The UIMS accepts and handles "asynchronous" messages that are sent by the NTM regarding the status of a request. The UI receives a message type 'SE' that is generated by an AP with the SIGERR NTM service call or by the NTM in its processing of a message in the service of an UI initiated AP. This message is sent to the UI only if the "test-mode" has been set to 1 or 2 by the UI or the error is designated as informational. The data returned to the UI includes the error code, severity level, name of the offending AP, and the error message. A "test-mode" of 2 limits the messages sent to the UI to fatal messages. After receiving an 'SE' message, the UI will display the message on the IISS terminal that initiated the request if "debug-mode" has been set by the user or if the error is designated as informational.

### 7.4.3 Test-mode And Debug-mode

The "test-mode" enabled switch currently depends on the role of the IISS user and the requested AP. The IISS user can set the "test-mode" on the function screen when initiating the AP. An enabled switch causes the UIMS to call the NTM services which tell the NTM that the "test-mode" is set. While in the test mode, if the IISS user hits the DEBUG key instead of the ENTER key, all error messages will be displayed on the terminal. This includes all UI messages and all NTM messages that are normally sent to the IISS operator's console. If the "test-mode" is not enabled, hitting the DEBUG key will have no effect at all and only informational messages will be displayed. If the "test-mode" is enabled but the DEBUG key is not hit, then only informational messages will be displayed.

### 7.4.4 Error Communication

The UI will use the NTM service SIGERR to inform the NTM of errors that will affect the future operation of the UI. The NTM can then display this error condition on the operator's console. If the UI is being run in stand-alone mode (not connected to the NTM), the error messages will be displayed on the attached terminal to ask the user to inform the IISS operator.

### 7.4.5 Message Management

The message management (MM) facility will be used to define and maintain the names and meanings of specific five character error codes. The files produced by MM will be used by the UI to display a meaningful error message on the IISS user's terminal based on the five character error code.

## 7.5 Common Data Model Processor

The CDMP software is divided into two categories, the precompiler that interacts with the user through the terminal (UI) and the query processor that interacts with the user through the user written application process.

All CDMP software will use ERRPRO to log all errors. It will also use SIGERR to send messages back to the user terminal upon encountering errors which affect the future operation of the software. These error indications will also be output on the operator's console. The errors will only be output on the user's terminal if "test-mode" and "debug-mode" are set.

### 7.5.1 Precompiler

The precompiler interacts with the user through a forms interface, thus allowing the precompiler to use the user's terminal as a means of displaying error messages. At termination time, one of the outputs generated from the precompiler is the listing file. This file contains all information about the syntax errors encountered during the precompile activity and the names of the AP's generated. The total number of syntax errors and the names of the generated APs will also be output to the IISS user's terminal.

On precompiler aborts such as from a bad input file name, the precompiler should terminate gracefully with an error message to the user terminal. The user may then reinitiate the precompile inputting the correct information.

### 7.5.2 Query Processing

The distributed request supervisor (DRS) is the management software for all data base queries. In order for the DRS to be informed of the unsuccessful starts of local request processors (LRP's), aggregators, and conceptual to external transformers and of other kinds of error conditions, the characteristics of the DRS must be set up to accept "asynchronous" messages from the NTM. Cognizance of these error conditions will prevent the DRS from waiting forever to get results.

The query processing is done as a result of a neutral data manipulation language (NDML) statement being encountered in an AP. A COBOL variable called NDML-STATUS has been reserved for the CDMP to return the status of the NDML data request to the AP. This allows the AP to differentiate between an error condition and the "no record found" condition. The error condition can then be recorded for further investigation and the "no record found" condition can indicate a legitimate answer for the data request.

## 7.6 COMMUNICATIONS SUBSYSTEM

COMM uses ERRPRO to log all errors to the error log file. Upon detecting a failure to communicate with the target host or other recoverable errors, COMM passes a message back to the NTM. The NTM then displays the message on the operator console so that the operator can investigate the problem and attempt to recover. An AP can be notified by the NTM of these communication problems through the use of "asynchronous" messages. This includes such errors as "binary message encountered when ASCII mode expected". The AP must be set up to accept these error messages or no error information is communicated.

## 7.7 IPC AND IHC

The IPC and IHC primitives call ERRPRO to log errors to the error log file. All errors, except the mailbox full condition, are logged as fatal errors. The primitives then return a non-zero status code to the calling routine so that appropriate action can be taken. There are no detectable errors that cause the primitives to abort.

✩U.S. GOVERNMENT PRINTING OFFICE: 1992 - 648-127/62309